

**SYBEX Sample Chapter**

# **.NET Framework Solutions: In Search of the Lost Win32 API**

**John Paul Mueller**

## **Chapter 5: Using Callback Functions**

Copyright © 2002 SYBEX Inc., 1151 Marina Village Parkway, Alameda, CA 94501. World rights reserved. No part of this publication may be stored in a retrieval system, transmitted, or reproduced in any way, including but not limited to photocopy, photograph, magnetic or other record, without the prior agreement and written permission of the publisher.

ISBN: 0-7821-4134-X

SYBEX and the SYBEX logo are either registered trademarks or trademarks of SYBEX Inc. in the USA and other countries.

TRADEMARKS: Sybex has attempted throughout this book to distinguish proprietary trademarks from descriptive terms by following the capitalization style used by the manufacturer. Copyrights and trademarks of all products and services listed or described herein are property of their respective owners and companies. All rules and laws pertaining to said copyrights and trademarks are inferred.

This document may contain images, text, trademarks, logos, and/or other material owned by third parties. All rights reserved. Such material may not be copied, distributed, transmitted, or stored without the express, prior, written consent of the owner.

The author and publisher have made their best efforts to prepare this book, and the content is based upon final release software whenever possible. Portions of the manuscript may be based upon pre-release versions supplied by software manufacturers. The author and the publisher make no representation or warranties of any kind with regard to the completeness or accuracy of the contents herein and accept no liability of any kind including but not limited to performance, merchantability, fitness for any particular purpose, or any losses or damages of any kind caused or alleged to be caused directly or indirectly from this book.

Sybex Inc.  
1151 Marina Village Parkway  
Alameda, CA 94501  
U.S.A.  
Phone: 510-523-8233  
[www.sybex.com](http://www.sybex.com)




## CHAPTER 5

---

# Using Callback Functions

---

- What Is a Callback Function?
  - Using Callback Functions
  - A Callback Function Example
- 

**C**hapter 4 provided you with a glimpse of some of the internals of the Win32 API. Message processing is a cornerstone of application development with the Win32 API, but it's only part of the equation. When an application sends a message, it hopes that another application will respond. Likewise, when an external application sends a message to your application, it's looking for a response. The problem is that this approach isn't two-way—it's a one-way communication from one application to another.

Callback functions provide the potential for two-way communication. When you make some calls to the Win32 API, you have to supply a pointer to a function that receives the response. This technique enables the Win32 API to provide two-way communication. A request from your application results in a response from the Win32 API to a specific point in your application. Two-way communication has important implications for the developer, as we'll discuss in this chapter.

After you gain an understanding of how callback functions work, we'll look at a callback function example. As you might imagine, getting callback functions to work under .NET is considerably more difficult than working in a pure unmanaged environment because you now have the managed environment to consider. It's not an impossible task, but there are certain restrictions you have to consider and a few programming techniques you'll want to learn.

**TIP**

Sometimes it's helpful to chat with other developers about questions you have in working with complex code. The VB World site at <http://www.vbforums.com/> offers both general and specific topic messaging areas. This site also offers general areas for discussions about other languages such as C#. VB World is exceptionally nice for those developers who prefer a Web interface to the usual newsgroup reader.

## What Is a Callback Function?

As previously mentioned, callback functions provide two-way communication. However, a callback function is more than a messaging technique—it's the Win32 API version of the asynchronous call. Your application makes a request and supplies the address of a callback function within your application. Windows will use this address as a communication point for the responses for your request. In many cases, Windows will call this function more than once—some callback functions are called once for each response that the Windows API provides.

Callback functions are important because they allow Windows to provide multiple responses for a single query. For example, when you want to scan the current directory on a hard drive, you actually need one response for each object in that directory. The same holds true for other response types. In this regard, you can view a callback function as a primitive form of collection. However, instead of gaining access to a single object that you have to parse one element at a time, the callback function provides individual elements from the outset.

**TIP**

We'll create more than a few callback functions as the book progresses. However, you might also want to view callback functions created by other developers. The Code Project includes a few examples of callback function coding on its site at <http://www.codeproject.com/win32/> and <http://www.codeproject.com/staticctrl/>. As mentioned on the page, many of these examples are unedited. Another interesting discussion appears on the C# Corner site at <http://www.c-sharpcorner.com/3/ExploringDelegatesFB002.asp>. I found this example a little convoluted, but some people may find it useful. The 4GuysFromRolla.com site at <http://4guysfromrolla.411asp.net/home/tutorial/specific/system/delegate?cob=4guysfromrolla> contains a number of interesting examples of both delegates and callback functions. Unfortunately, some of the code is also based on Beta 1 of Visual Studio .NET, so you'll need to select examples with care.

Most callback functions have a specific format because you need to know specifics about the object, such as the object type. The use of a specific format also provides a standard communication format between the Win32 API and the requesting application. The message format provides a means of passing information in a specific manner between the Win32 API and the calling application.

In many cases, a callback function can also provide feedback to the message sender. For example, you might not want to know the names of all of the files in a directory—you might only need one file. Once the application finds what it needs, it can tell the Win32 API to stop sending information. We'll see this particular feature in many applications in the book, even the MMC snap-in example in Chapter 12.

Like messages, the .NET Framework also has to provide support for callback functions. However, in this case you can't interact with the callback function directly. What you see instead is a collection that contains the requested data. In most cases, this loss of intermediate result control is a non-issue. There are a few situations, such as a file search, when you can gain a slight performance boost using an actual callback function. In general though, you should only rely on callback functions when the .NET Framework doesn't provide the desired functionality.

## Using Callback Functions

Now that you have a better idea of what a callback function is and how you'd use it, let's look at some practical issues for using callback functions. The following sections describe the callback function prototypes and essential design techniques. You'll learn about callback function design using a simple example.

The point of this section is to provide you with a template that you can use in developing other types of callback functions for your applications. The essential task list remains the same, even when the callback function you use changes. For example, you'll always use a delegate to provide a callback address for the Win32 API function—no matter how complex the Win32 API function is or what task it ultimately performs.

## An Overview of Callback Function Prototypes

Callback functions are unique, in some respects, because they provide a feedback method from Windows to the application. To ensure that Windows and the callback function use the same calling syntax (a requirement for communication), the Platform SDK documentation provides a set of callback function prototypes—essentially a description of the callback function argument list.

---

**NOTE**

This chapter doesn't discuss the special callback function prototypes for DirectX. For a discussion of DirectX callback function prototypes, see the DirectX Callback Function Prototypes section of Chapter 14. In many ways, the DirectX callback prototypes look and act the same as the prototypes in this chapter. However, the calling syntax is quite specific, so you need to know more about them before working with DirectX in applications.

When you make a system request that includes a callback function, you need to supply the address of the callback function matching the function prototype for that call. For example, the EnumWindows() and EnumDesktopWindows() functions both use the same function prototype in the form of the EnumWindowsProc() shown in the following code.

```
BOOL CALLBACK EnumWindowsProc
(
    HWND    hwnd,    // handle to parent window
    LPARAM  lParam    // application-defined value
);
```

In order to use either the EnumWindows() or the EnumDesktopWindows() function, you must provide the address of a prototype function that includes the handle to a parent window and an application-defined value. The prototypes for other callback functions are all standardized, but vary according to the Win32 API call that you make. It's important to research the callback function to ensure you supply one with the proper arguments in the right order.

---

**TIP**

Arguments for callback functions follow the same rules as function and message calls. For example, you'll still use an IntPtr for a handle. It pays to check the argument list carefully so that you can avoid defining application-supplied and -reserved arguments incorrectly.

Unfortunately, the prototype description won't tell you about the purpose of the application-defined value. To learn about the application-defined value, you need to look at the documentation for the individual functions. In the case of `EnumWindows()` and `EnumDesktopWindows()`, you don't receive any additional information from the application-defined value unless that information is passed as part of the original call.

The only piece of information your callback function will receive from the `EnumWindows()` function is a handle to the window. The function will continue to call your callback function with one window handle at a time until your callback function returns false (indicating you don't need any more data) or the function runs out of handles to return. You can use the window handle in a number of ways. For example, you could send the window a message as we did in Chapter 4. However, there are a number of other window-related functions that have nothing to do with messaging—you could simply learn more about the window using the `GetWindowText()` or `GetWindowInfo()` functions.

## Implementing a Callback from the Managed Environment

It's time to look at the first callback example. This example is designed to break the callback creation process down into several discrete steps. In this case, we'll discuss what you need to do to enumerate the current windows. Enumerating the windows is the first step in discovering windows that you might want to communicate with—an important part of the messaging process. The source code for the example appears in the `\Chapter 05\C#\EnumWindows` and `\Chapter 05\VB\EnumWindows` folders of the CD.

### Creating a Delegate

The first task you need to perform in creating a callback function is to define a delegate to represent the function. You can't pass the address of a managed function to the unmanaged environment and expect it to work. The delegate provides the means for creating a pointer that CLR can marshal. We'll see as the example progresses that the delegate is easy to use but important in effect.

---

**TIP**

In general, you'll use an event setup (as shown in Chapter 4) to handle Windows messages. However, you'll use delegates to enable use of callbacks. The main reason you want to use events to handle Windows messages is to allow someone inheriting from your code to access the message without worrying about the details of the Windows message. In addition, this technique works better where multiple threads are involved. Make sure you check thread safety when handling both Windows messages and callbacks. Normally, thread safety is less of a concern when handling callbacks, so the delegate technique shown in this chapter works fine.

The delegate you create must match the callback function prototype. In fact, giving the delegate the same name as the prototype helps document your code for other developers. Notice that the delegate shown in the following requires an `IntPtr` for the window handle and an `Int32` for the `lParam`.

```
// Create the delegate used as an address for the callback
// function.
public delegate bool EnumWindowProc(IntPtr hWnd, Int32 lParam);
```

### Creating the Callback Function

The callback function performs the actual processing of the data returned by the call to the Win32 API function. The main thread of your application will go on to perform other tasks while the callback function waits for data. Listing 5.1 shows the callback function used for this example.



#### Listing 5.1 Creating the Callback Function

```
// Define a function for retrieving the window title.
[DllImport("User32.DLL")]
public static extern Int32 GetWindowText(IntPtr hWnd,
                                         StringBuilder lpString,
                                         Int32 nMaxCount);

// Create the callback function using the EnumWindowProc()
// delegate.
public bool WindowCallback(IntPtr hWnd, Int32 lParam)
{
    // Name of the window.
    StringBuilder TitleText = new StringBuilder(256);

    // Result string.
    String ResultText;

    try
    {
        // Get the window title.
        GetWindowText(hWnd, TitleText, 256);
    }
    catch (Exception e)
    {
        MessageBox.Show("GetWindowText() Error:\r\n" +
                        "\r\nMessage: " + e.Message +
                        "\r\nSource: " + e.Source +
                        "\r\nTarget Site: " + e.TargetSite +
                        "\r\nStack Trace: " + e.StackTrace,
                        "Application Error",
                        MessageBoxButtons.OK,
                        MessageBoxIcon.Error);
    }
}
```

```
// See if the window has a title.
if (TitleText.ToString() == "")
    ResultText = "No Window Title";
else
    ResultText = TitleText.ToString();

// Add the window title to the listbox.
txtWindows.Text += ResultText + "\r\n";

// Tell Windows we want more window titles.
return true;
}
```

As you can see, the `WindowCallback()` relies on the `GetWindowText()` function to display the name of the window in a textbox on the dialog. The use of an `IntPtr` as one of the inputs is hardly surprising, because it contains the handle to the window pass to `WindowCallback()` by Windows. Remember that in the past we always used a `String` to pass text data to the Win32 API function. The `GetWindowText()` function requires a different technique, however, because it actually creates the string—it allocates the memory for the string and places the data in it. Using a `StringBuilder` object enables the `GetWindowText()` function to behave as normal. If you try to use a standard `String` in this case (even one passed with the `out` or `ref` keyword) the function call will fail and the user will see an error on screen.

Notice that the use of a `StringBuilder` object becomes clearer in the `WindowCallback()` function. The code allocates a `StringBuilder` object of a specific size. It then passes this size to the `GetWindowText()` function in the third argument, *nMaxCount*.

**WARNING** Depending on how you set up your callback function, it's possible that the callback function will operate in a different thread from the main form. When the callback function operates in a separate thread, it can't change the content of the main form; otherwise, you might run into thread-related problems with the applications (see the "Developing for Thread Safety" section of Chapter 4 for details). It pays to validate your application for thread safety by viewing the callback function in the debugger using the Threads window. If you see that the application creates a new thread, then you'll need to use an event to trigger changes to the display area.

Always place the `GetWindowText()` and other string manipulation functions within a `try...catch` block as shown in the code. These functions tend to fail, at times, even if your code is correct. Unfortunately, there isn't any documented reason for the failure and it occurs intermittently—making the cause exceptionally difficult to track down. The example code shows the minimum error message you should provide as output if the `GetWindowText()` call fails. You might consider checking the inner error messages as well as using the `GetLastError()` function to return any Windows-specific information about the error.



A successful call to `GetWindowText()` is no guarantee that *TitleText* will contain any data on return from the call. In fact, you'll find that many of the hidden windows have no title bar text at all, which means that `GetWindowText()` will return an empty string. With this in mind, you'll want to create a standard string and place either a default value or the contents of *TitleText* within it. *ResultText* contains the string that we'll actually display on screen. The display code is straightforward—you simply add to the text already found in the textbox.

Notice that `GetWindowText()` always returns a value of *true*. Because we want the name of every window on the desktop, you have to keep returning *true*. However, not every callback function will require all of the data that Windows has to provide. If this is the case, you'll want to add an end of data check and return *false* if the function has all of the data it needs.

### Demonstrating the EnumWindows() and EnumDesktopWindows() Callback Functions

At this point, you have a delegate to provide a pointer to the callback function and a callback function to process the data. All you need is some way to call the Win32 API function with the callback function as one of the arguments. Listing 5.2 shows how to accomplish this task.



#### Listing 5.2 Code for Enumerating all Windows or a Single Desktop

```
// Create the prototype for the EnumDesktopWindows() function.
[DllImport("User32.DLL")]
public static extern void EnumDesktopWindows(IntPtr hDesktop,
                                           EnumWindowProc EWP,
                                           Int32 lParam);

// Create the prototype for the EnumWindows() function.
[DllImport("User32.DLL")]
public static extern void EnumWindows(EnumWindowProc EWP,
                                      Int32 lParam);

private void btnTest_Click(object sender, System.EventArgs e)
{
    // Create an instance of the callback.
    EnumWindowProc PWC = new EnumWindowProc(WindowCallback);

    // Clear the text window.
    txtWindows.Clear();

    // Call the EnumWindows() function.
    EnumWindows(PWC, 0);
}

private void btnTest2_Click(object sender, System.EventArgs e)
{
    // Create an instance of the callback.
    EnumWindowProc PWC = new EnumWindowProc(WindowCallback);
```

```
// Clear the text window.
txtWindows.Clear();

// Call the EnumDesktopWindows() function.
EnumDesktopWindows(IntPtr.Zero, PWC, 0);
}
```

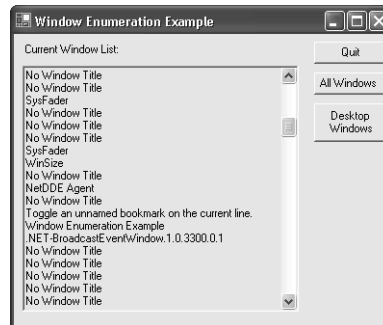
The example provides both a general and a desktop-specific version of the windows enumeration functions, `EnumDesktopWindows()` and `EnumWindows()`. Notice that the `EnumDesktopWindows()` function prototype uses an `IntPtr` for the window handle as usual. However, the callback function pointer is marked as the `EnumWindowsProc` delegate. This isn't an error—you actually pass the delegate as a pointer in the code. The final argument is the *lParam* that you can use for application-specific data (we won't for this example).

Look at the `btnTest_Click()` and `btnTest2_Click()` methods. The first method is used for general windows enumeration, while the second is used for desktop-specific enumeration. Both follow the same sequence of steps to gain access to the appropriate Win32 API function.

The code begins by creating an instance of the `EnumWindowProc` delegate with the `WindowCallback()` function as a pointer. The code clears the textbox so you don't see the previous data. It then calls the appropriate windows enumeration function. When you run this code, you'll see that the Win32 API begins sending the callback function data almost immediately. Figure 5.1 shows the results.

**FIGURE 5.1:**

The test application shows a complete list of windows for the system.



It shouldn't be too surprising that there are a lot of unnamed windows listed in the example. Windows constantly creates hidden windows that perform tasks silently in the background. However, looking through the list of windows that do have names can prove interesting. For example, the example detected a previously unknown ".NET-BroadcastEventWindow.1.0.3300.0.1" window.

The point is that you can list the windows as needed. Other functions, such as the `GetTitleBar()` function provide more information about each window, including the presence and use of various common buttons. For example, you'd use the `GetTitleBar()` function to determine if the window in question has a functional Minimize button. The more generic `GetWindow-Info()` function tells you about the window's features and setup. For example, you can determine the location and size of the window, as well as its style information.

## Implementing a Callback from a Wrapper DLL

There are going to be times when you use a callback function so often that placing it into each of your applications individually doesn't make sense. However, creating a lot of duplicate code isn't the only reason to use the wrapper DLL. The following list provides some additional reasons you should use this technique in your next application.

**Packaging Issues** Using a wrapper DLL enables you to package the calling details in a way that you can't do normally. Using a DLL becomes a matter of convenience because the developer sees a package, not lines of code. In addition, when you work with a team of developers, you might want to hide the details of the Win32 API call to make the function easier to use.

**Team Development Issues** The biggest advantage for a team is that one group of developers can work on Win32 API calls while other groups work on application code. The use of a DLL detaches one effort from the other and allows both groups to work independently. In addition, because everyone's using the same DLL, you can ensure better consistency among developers, making the resulting code easier to read.

**Learning Curve and Training Issues** Another advantage is learning curve. Many of the developers working on a team will know their base language well, but won't know much about the Win32 API, so trying to get them up to speed represents a significant training cost. Having a team that specializes in making the Win32 API fully accessible to other members on your team makes sense because Microsoft will almost certainly fill many of the holes in the next version of Visual Studio. (It's unlikely that Microsoft will ever fill all of the holes, which means you'll always need someone who can work with the Win32 API.)

The example in this section duplicates the functionality of the `EnumWindows` example presented earlier in the chapter. However, instead of placing all of the Win32 API code within the dialog-based application, it will appear within a wrapper DLL. The dialog-based application will see a collection in place of the Windows-specific data. The example serves to demonstrate two elements of using a wrapper DLL.

- The initial development effort is harder because you need to write more code and the wrapper DLL code has to interact with the application.

- Using the DLL in subsequent development efforts is easier than including the Win32 API code, because the developer need not understand the Win32 API to make the required call.

### Creating the Library DLL

The first step in creating this example is to create the wrapper DLL. For the purposes of the example, the wrapper DLL and dialog-based application appear in the same folder on the CD, but you could easily place each element in a separate folder. Listing 5.3 contains the DLL code for the example. You'll find the source code for this example in the \Chapter 05\C#\LibraryAccess and the \Chapter 05\VB\LibraryAccess folders of the CD.

#### NOTE

Listing 5.3 contains only the code for the EnumWindows() function. The EnumDesktopWindows() function code is essentially the same. You can see the minor differences by looking at the source code on the CD.



#### Listing 5.3

#### The DLL Contains All the Win32 API Calls and Returns a Collection

```
public class AllWindowCollection : CollectionBase
{
    // We could place the code for calling the windows enumerator
    // in the constructor, but using the Fill() function adds more
    // control and becomes important in the DesktopWindowCollection
    // class.
    public AllWindowCollection()
    {
    }

    // Create the delegate used as an address for the callback
    // function.
    private delegate bool EnumWindowProc(IntPtr hWnd, Int32 lParam);

    // Create the prototype for the EnumWindows() function.
    [DllImport("User32.DLL")]
    private static extern void EnumWindows(EnumWindowProc EWP,
                                           Int32 lParam);

    // Fills the collection with data you can access using the
    // Item() function.
    public void Fill()
    {
        // Create an instance of the callback.
        EnumWindowProc PWC = new EnumWindowProc(WindowCallback);

        // Call the EnumWindows() function.
        EnumWindows(PWC, 0);
    }
}
```

```

// Obtains a specific window title string from the collection
// and returns it to the caller.
public string Item(int Index)
{
    return (string)List[Index];
}

// Define a function for retrieving the window title.
[DllImport("User32.DLL")]
private static extern Int32 GetWindowText(IntPtr hWnd,
                                           StringBuilder lpString,
                                           Int32 nMaxCount);

// Create the callback function using the EnumWindowProc()
// delegate.
private bool WindowCallback(IntPtr hWnd, Int32 lParam)
{
    // Name of the window.
    StringBuilder TitleText = new StringBuilder(256);

    try
    {
        // Get the window title.
        GetWindowText(hWnd, TitleText, 256);
    }
    catch (Exception e)
    {
        // Throw an exception when required.
        throw new Exception("Error Accessing Window Titles", e);
    }

    // See if the window has a title.
    if (TitleText.ToString() == "")
        List.Add("No Window Title");
    else
        List.Add(TitleText.ToString());

    // Tell Windows we want more window titles.
    return true;
}
}

```

---

Listing 5.3 shows that there are some differences between a wrapper DLL version of a Win32 API call and the application version. (There are also many similarities between the two implementations—you still need to perform the same set of tasks as before.) Notice that all of the Win32 API calls are declared private, to hide them from view and protect their functionality. In addition, this class inherits from the `CollectionBase` class, so it already has much of the functionality required for a collection.

The `Fill()` function is new. It takes the place of the `btnTest_Click()` function in the previous example. However, notice that this function never touches the form objects, so you don't have to worry about thread concerns. The `Fill()` function is also simpler than the `btnTest_Click()` function—not that complexity was a problem with the previous example.

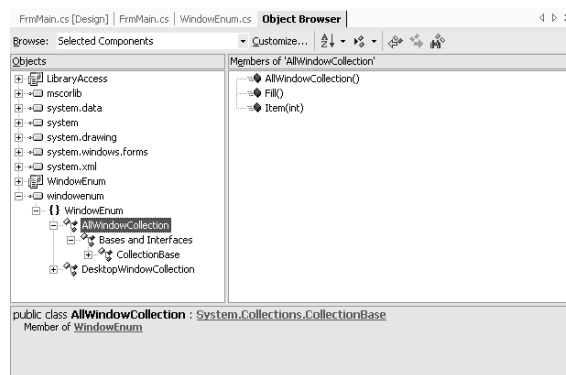
You also have to include an `Item()` function with the collection so that the user can gain access to the collection elements. You can make this function as simple or complex as you like. The example shows a basic implementation that returns the requested element from the `List` object inherited from the `CollectionBase` class. One of the additions you might want to make is a range check to ensure the input isn't out of range.

The `WindowCallback()` has changed from the previous example. For one thing, the `try...catch` block throws an exception now instead of displaying an error message. Using this approach ensures that the developer using your library has full access to all of the error information from the call. Another change is that we're adding items to the `List` object now instead of creating the output directly. Again, this change ensures there are no threading problems with the application because the callback function isn't touching any of the form objects.

The biggest change is simplicity for the developer using the new library. Figure 5.2 shows the Object Browser view of the library. Notice that the interface is exceptionally simple—most of the functionality appears within the `CollectionBase` class and isn't even implemented in your code. Any developer who's worked with collections in the past will understand how your collection works as well. A simple interface combined with common usage techniques makes the library approach hard to beat in this case. Of course, you do have to perform additional work at the outset, which can be viewed as a disadvantage.

**FIGURE 5.2:**

The Object Browser view says it all—libraries make Win32 API calls easy to use.



## Creating the Dialog-Based Application

Once you create a wrapper DLL for the Win32 API calls, creating the application to use the functionality that the wrapper DLL provides is relatively simple. The example uses a collection to hold the information gathered by the Win32 API call, so you'll create a function to access the collection as shown in Listing 5.4.

---

**NOTE**

Listing 5.4 contains only the code for the `btnTest_Click()` function. The `btnTest2_Click()` function code is essentially the same. You can see the minor differences by looking at the source code on the CD.



---

**Listing 5.4      The Dialog-Based Application Code Looks Like Any C# Code**

```
private void btnTest_Click(object sender, System.EventArgs e)
{
    // Create a StringBuilder object to hold the window strings.
    StringBuilder WindowList = new StringBuilder();

    // Create an instance of the collection.
    AllWindowCollection AWC = new AllWindowCollection();

    // Fill the collection with data.
    AWC.Fill();

    // Clear the textbox contents.
    txtWindows.Clear();

    // Create a single string with the contents of the collection.
    for (int Counter = 0; Counter < AWC.Count; Counter++)
        WindowList.Append(AWC.Item(Counter) + "\r\n");

    // Display the string on screen.
    txtWindows.Text = WindowList.ToString();
}
```

---

This code makes some improvements over the previous example and you'll likely notice the difference when you use this function with a lot of windows open. The `StringBuilder` object, `WindowList`, provides a significant performance boost because you don't have to rebuild the string for every collection entry. A `StringBuilder` uses the `Append()` function to add new strings to the contents of the object. You'll find that using a `StringBuilder` also saves resources because the code isn't creating a new string for every iteration of the for loop.

Instead of worrying about Win32 API functions, the example creates the `AllWindowCollection` object. If you look at the functions provided by this object, you'll see a list that

combines the custom functions we created with a list of generalized collection functions. For example, you can use the `Clear()` function to empty the collection, even though that function isn't implemented in the custom code.

The code calls the `Fill()` function to fill the collection object, `AWC`, with data. This function is all that the developer using the wrapper DLL needs to know in order to make the Win32 API calls discussed earlier. When the call returns, `AWC` contains a complete list of the window titles for the current machine.

The next step is to place the formatted string into `WindowList`. The example uses all of the strings, but you can easily filter the strings because we're using a collection. For that matter, you can also sort the strings and perform other tasks that the initial example code can't do with any ease. Notice that `AWC` has a `Count` property that makes iterating through the items in the collection easy.

The final step is to place the string into the textbox. Notice that we have to use the `ToString()` function because C# views the `StringBuilder` object as something other than a string reference. The output of this example is precisely the same as the output of the first example. You'll see a display that looks like the one shown in Figure 5.1.

## Enumerating Calendar Information Example

The .NET Framework provides a vast array of classes for handling international information. You'll find them in the `System.Globalization` namespace. There's so much functionality that sometimes it's hard to find precisely what you need. However, even given the rich array of functions that the .NET Framework provides, there are still times when you need a simple way to list information about a culture. For example, what does a particular culture call the days of the week or the months of the year? The example in this section of the chapter is meant to augment what the .NET Framework already provides. (The fact is that the .NET Framework provides far better functionality overall than the Win32 API in this case.)

This example also brings up a new topic: what do you do with macros? Visual C++ developers have long been familiar with the functionality provided by macros, something that other languages don't support very well without a lot of work. There are two ways to handle the macros. You can create a Visual C++ wrapper and call the macro directly, or you can simulate the macro using managed code. Generally, you'll find that the Visual C++ wrapper method is easier and less error prone, so that's the method we'll use in this example.



**TIP**

Microsoft has made a wealth of .NET training information available through the Microsoft Developer Network (MSDN) Academic Alliance (MSDNAA) site (<http://www.msdnaa.net/technologies/dotnet.asp>). Make sure you spend some time at this site looking through the offerings—including those that relate to delegates and callback functions.

Now that you have some idea of what this example will show, let's look at some source code. The following sections tackle the various problems of enumerating calendar values using a Win32 API function with callback. You'll find the source code for this example in the \Chapter 05\C#\CalendarCheck and \Chapter 05\VB\CalendarCheck folders of the CD. The macro wrapper DLL source code is located in the \Chapter 05\C#\CalendarCheck\Locale-Macros folder—you can use the same DLL for both versions of the example.

## Creating the Macro Wrapper DLL

Visual C++ includes a number of macros used to convert one type of input into another type of input. In many cases, the macro converts two values into a single long value. For example, the macro might convert two WORD values into a single DWORD value with the first WORD located in the high WORD of the DWORD and the second WORD loaded in the low WORD of the DWORD. Modern code doesn't use this technique, but it was quite common when Windows first arrived on the scene, so we still have to contend with this method of transferring data today.

**WARNING**

You must include `Windows.H` as part of `STDAFX.H` to make most Visual C++ wrapper DLLs work correctly. In addition, you must include certain `#defines` to ensure that the compiler will enable advanced Windows features. The `STDAFX.H` entries provided with the example code show the most common additions. We'll see later in the book that this is a baseline configuration. For example, if you want to create an MMC Snap-in, you also need to include `MMC.H` in `STDAFX.H`. The order of the `#includes` is important—placing an `#include` in the wrong place can cause the code to compile incorrectly or not at all.

Listing 5.5 shows the code you'll need to use the `MAKELANGID()` and `MAKELCID()` macros. Don't confuse macros with functions—they're not interchangeable. You'll always need to create a Visual C++ wrapper DLL to use a macro, but most functions are easily accessible from within the .NET host language.



### Listing 5.5 Macro Wrapper for Locale Conversion

```
// Create a language ID to use with the DoMAKELCID() function.
static Int16 DoMAKELANGID(Int16 usPrimaryLanguage, Int16 usSubLanguage)
{
    return MAKELANGID(usPrimaryLanguage, usSubLanguage);
}
```

```
// Create a LCID to use with functions like EnumCalendarInfoEx().
static Int32 DoMAKELCID(Int16 wLanguageID, Int16 wSortID)
{
    return MAKELCID(wLanguageID, wSortID);
}

// Convenient way to obtain the LOCALE_SYSTEM_DEFAULT value.
static Int32 GetLocaleSystemDefault()
{
    return LOCALE_SYSTEM_DEFAULT;
}

// Convenient way to obtain the LOCALE_USER_DEFAULT value.
static Int32 GetLocaleUserDefault()
{
    return LOCALE_USER_DEFAULT;
}
```

As you can see, the `DoMAKELANGID()` and `DoMAKELCID()` functions simply transfer the incoming data to the macros and then return the result. Some macros require data conversion and a few can get quite complex. However, this code represents the vast majority of the macro conversions that you'll perform. The only reason you need to use Visual C++ at all is to access the macro.

**NOTE**

The source code found in this section of the chapter is smaller than what you'll find on the CD. The macros and the `EnumCalendarInfoEx()` function both require enumerations to ensure the data input is correct. Because there isn't anything interesting about the enumerations (other than their presence), the code in the book only contains the actual methods.

There are many situations where you'll see default values listed in the Platform SDK documentation that are actually macro results. In many cases, you can duplicate the default values in your code, but it's just as easy to request the default value from Visual C++. Never assign a constant value to a default value derived from a macro because the macro inputs could change. The `GetLocaleSystemDefault()` and `GetLocaleUserDefault()` obtain the two default values for this example from Visual C++. We'll see in the "Demonstrating the Calendar Enumeration" section how to perform this same task using the in code method.

**TIP**

If you're finding the new Visual C++ .NET Managed Extensions difficult to figure out, Microsoft provides an instructor-led course (2558) that covers this particular part of the product in detail. You can learn more at <http://www.microsoft.com/TRAINCERT/SYLLABI/2558APRELIM.ASP>.

One of the issues you need to work around is the oddity of working with Visual C++ in the managed environment. This often means changing your coding style or becoming aware of a new code word. In the case of enumerations, you need to add a `__value` keyword as shown here.

```
__value enum SortID
{
    SI_DEFAULT          = 0x0,      // sorting default
    // Some skipped values here...
    SI_GEORGIAN_MODERN = 0x1      // Georgian Modern order
};
```

Adding the `__value` keyword will change the presentation of the enumeration within Visual C++. The symbol will change to show that this is a managed enumeration as shown in Figure 5.3. Notice that the enumeration is also part of the class and doesn't simply exist in the namespace. The code will compile if you place the enumeration in the namespace without a class, but you won't be able to see it when you import the DLL into another language (as we will for the example). Another point of interest is that the Object Viewer will display your comments as long as you're looking at the Visual C++ view of the enumeration.

**FIGURE 5.3:**

Using the `__value` keyword changes the presentation of the enumeration in Visual C++.

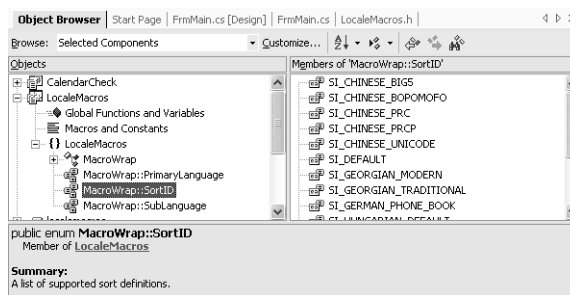
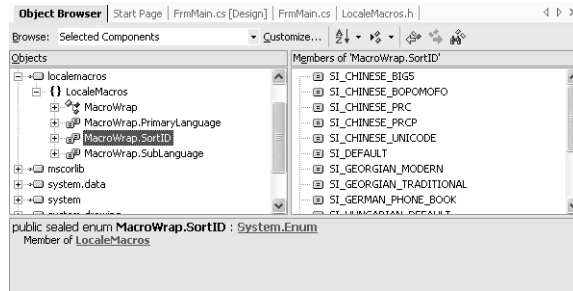


Figure 5.4 shows the imported view of the same DLL shown in Figure 5.3. The first thing you should notice is that the enumeration now uses the standard symbol, as if we hadn't done anything special in Visual C++. This is an important piece of information to remember, because it demonstrates that the viewing DLLs in the Object Viewer will tell you about the content of the DLL but not necessarily about the tricks used to produce that content.

You should also notice the lack of comments in Figure 5.4. Even though you can see the comments in the Visual C++ presentation, you won't see them when the DLL is imported into another language. Unfortunately, there isn't a fix for this problem unless you want to resort to some truly interesting coding in the CLR intermediate language (IL). The best way around this problem for now is to ensure that your Visual C++ function names are clear, conform to any Windows documentation the user might already know, and follow any documentation you create for the DLL.

**FIGURE 5.4:**

Even though a Visual C++ enumeration requires special handling, the Object Viewer won't show it.



## Creating the EnumCalendarInfoEx() Function Code

As in the previous examples, one of the first steps in using a callback function is to create a delegate and a callback function to handle the input. The delegate and callback functions for this example rely on the EnumCalendarInfoProcEx() prototype found in the Platform SDK documentation. Listing 5.6 shows both of these elements.



### Listing 5.6 Creating a Delegate and Callback Function for EnumCalendarInfoEx()

```
// Create the delegate used as an address for the callback
// function.
public delegate bool EnumCalendarInfoProcEx(
    String lpCalendarInfoString,
    CALID Calendar);

// Create the callback function.
public bool CalendarCallback(String lpCalendarInfoString,
    CALID Calendar)
{
    // Create the output string.
    txtCalOutput.Text = txtCalOutput.Text +
        Calendar + "\r\n" +
        lpCalendarInfoString + "\r\n\r\n";

    // Make sure we return all of the values.
    return true;
}
```

Notice that that callback function receives two inputs. The first is the information string that the caller requested. The second is a calendar identifier that tells which calendar reference the string is using. By using an enumerated type as input, rather than an Int32, the code saves a little work. You can place the returned enumerated value directly in the output string and C# won't complain. We'll see later how this works.

## Demonstrating the Calendar Enumeration

One of the most important things to remember about the `EnumCalendarInfoEx()` function is that it's machine specific. You can only list information for the languages actually installed on the machine. If you don't know which languages the machine has installed, then it's usually safer to ask for an enumeration of all languages. Enumerating all of the languages when there's only one language installed won't produce a different result from asking for the specific language—it simply frees you from finding out which language is installed on the machine.

This example can return quite a few different types of data and it's interesting to view them all. Consequently, the example provides a drop-down list box you can use to select information of interest. Clicking Test will display the data. Listing 5.7 shows the source code for demonstrating the `EnumCalendarInfoEx()` function.



### Listing 5.7 Demonstrating the EnumCalendarInfoEx() Function

```
// Retrieves the requested calendar values.
[DllImport("Kernel32.DLL")]
public static extern void EnumCalendarInfoEx(
    EnumCalendarInfoProcEx pCalInfoEnumProcEx,
    Int32 Locale,
    CALID Calendar,
    CALTYPE CalType);

private void btnTest_Click(object sender, System.EventArgs e)
{
    // Create the callback pointer.
    EnumCalendarInfoProcEx ECIPE = new EnumCalendarInfoProcEx(CalendarCallback);

    // Create the language ID.
    Int16 LANG_SYSTEM_DEFAULT = MacroWrap.DoMAKELANGID(
        (Int16)MacroWrap.PrimaryLanguage.PL_NEUTRAL,
        (Int16)MacroWrap.SubLanguage.SL_SYS_DEFAULT);

    // Create the LCID.
    Int32 Locale = MacroWrap.DoMAKELCID(
        LANG_SYSTEM_DEFAULT,
        (Int16)MacroWrap.SortID.SI_DEFAULT);

    // Clear the textbox.
    txtCalOutput.Clear();

    // Call the calendar enumeration.
    EnumCalendarInfoEx(ECIPE,
        Locale,
        CALID.ENUM_ALL_CALENDARS,
        (CALTYPE)cbCalSelect.SelectedIndex+1);
}
```

---

The example begins by creating a callback pointer. You can consider this the first step in working with any callback function. Make sure you always use the delegate to create the pointer or the code won't work.

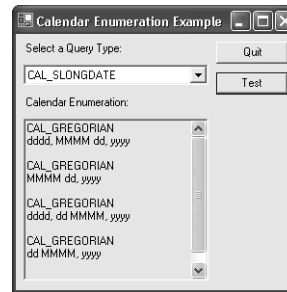
The next two calls create the locale identifier (LCID). The first, `DoMAKELANGID()`, accepts a primary and secondary language as input. The numbers for these inputs are provided as part of enumerations in the source code. You'll want to do the same thing, whenever possible, to ensure the input to the macros is always correct. (They won't ever report an error, so debugging this kind of problem is frustrating, to say the least.)

The second call, `DoMAKELCID()` accepts the language identifier created in the first step, along with a sort order. Again, this is an enumeration based on the contents of the C/C++ header files provided with Visual Studio .NET. The return value of this second step is the LCID that you need for the `EnumCalendarInfoEx()` call.

The final two steps are to clear the contents of the textbox (`txtCalOutput`) and call the enumerator `EnumCalendarInfoEx()`. One of the essentials here is to ensure any data conversions are correct, which is why one of the arguments, `(CALTYPE)cbCalSelect.SelectedIndex+1`, contains a typecast and I've increased it by one. Figure 5.5 shows the output of this example.

**FIGURE 5.5:**

The example provides information about the language installed on the current machine.



## Where Do You Go from Here?

Callback functions are an essential part of using the Win32 API. You won't need to use them as often as other tricks of the trade under .NET, but you'll need them just the same. This chapter has helped you understand what a callback function is, how and when to use it, and demonstrated the kinds of applications you can create using a callback function. However, there are still many issues to discuss for callback functions, so we'll look at this topic again as the book progresses.

Now that you have some idea of what a callback function is and where you'll commonly use it, it's time to look at some .NET code. Look for places where you suspect a collection is

really standing in for a callback function found in the Win32 API. You might be surprised at the amount of overlap that you see. Make sure you check out some of the example sites listed in the chapter as well. It's always interesting to see how someone else would tackle the problems of working with callback functions.

It's also important to consider how you might use callback functions in combination with wrapper DLLs. In some cases, you'll want to handle most of a call using unmanaged code to prevent the performance-robbing cost of switching between the managed and unmanaged environment. Using a callback function could help you gain a modicum of flexibility over a function that normally returns more than one result, while reducing the performance overhead of interacting with the DLL.

Chapter 6 begins a new phase of this book. Rather than look at the technologies involved in working with the Win32 API, we'll start seeing how you can put the information learned so far to work. One of the common places to use the Win32 API is at the console screen. The .NET Framework lacks functionality in this area now because it's not one of the areas that Microsoft targeted during development. Chapter 6 will show you some ways to enhance your console applications and provide the user with a better experience—while you gain the benefits of using .NET for your application development.